New Feature

For the implementation of a new feature I chose to implement a watchlist. This watchlist is only displayed when a user is logged in. A user can navigate to a movie page by either browsing through the movies by their first letter or by using the search functionality.

Once the user is at the movie page and is logged in, they will have an “Add to Watchlist” button on their screen. If the user presses the button, the page will refresh, and the sidebar will now contain their movie in their watchlist.

When the user leaves the movie page, the movie in their watchlist is always displayed in the sidebar. The user can click on the movie title in the watchlist to navigate back to the relevant movie.

When the movie is in the watchlist, if the user navigates back to the movie, instead of a “Add to Watchlist” button the user will see a “Remove from Watchlist” button. If the user presses this button, the page will refresh, and the watchlist will no longer contain the movie, and the movie will no longer be displayed on the sidebar.

The Watchlist is a property of the User object and therefore ensures that the watchlist is associated with the user.

Key Design Decisions

Domain Model

A key design decision that I made was the creation of a make_review function. This function makes sure that the links between user, movie and the review from the user are properly connected at the start. A review for the purposes of the application always needed to be linked to a user and to a movie and vice versa so that it could be properly managed for the web app. The use of the function guarantees this and doesn’t create some form of mismanagement of a review that a user leaves. Without this function a review could be incorrectly made without the link to a user and to the movie.

Repository

A key design decision for the applications repository was the use of the Repository pattern. The use of this pattern creates dependency inversion. This means that the service layer of the application doesn’t directly rely on the memory repository. Instead both the service layer and the memory repository rely on abstractions and in particular the abstract repository interface. The advantage of the service layer relying on the abstract repository interface is that the memory repository can be used when first implementing the web app and then later on the memory repository can be switched with a database repository that

stores data persistently. Because the service layer relies on the abstract interface the switch can be made without having to make any changes to the service layer. The web app can then run with a database with the only minimal changes needing to be made.

Blueprints

Blueprints where used to separate the responsibility for certain areas of the application. The blueprints were separated into: movies which is responsible to displaying movie information, authentication which is responsible for registering a user with the web app as well as the logging in and out of the user, and lastly home which was responsible for displaying the home page. This separation of responsibility helped the application adhere to the principal of Single Responsibility. This means that every part of the application had responsibility for a single piece of the program's functionality. The use of the single responsibility principle helped me understand and develop parts of the application faster.

For example, when I was implementing the view and service layer for movies it was easier to write unit tests when they only effected that part of the application. This allowed me to write that element faster. It also allowed me to use and interact with the web app before I implemented the authentication part of the web app as the movies part of the program didn't involve the authentication part. It also allowed for a more iterative approach to development where I could test and implement small components one at a time.

Search

During the implementation of the search feature I create a function called elements_in_common inside of the movies services layer. This function's original purpose was to find the movies that were apart of any combination of genre, actor, and director. I implemented this function with the idea of extensions in mind. I created this function so it would be able to find the intersection between any variable number of lists containing any type of object. Implementing this function in this way leaves it open to extension in other parts of the search functionality. The search function could be adapted to search for movies with any number of restrictions e.g. the starting letter of the movie, the genre, the year etc. and the elements_in_common function would not need to be adapted as it is not reliant on the number of lists or the type of objects in those lists.

Templating

When creating the layout of the web app I decided on all the key features that I wanted to be consistent across all the webpages. I then created a layout.html page that contained these consistent features, that all of my webpages could inherit. The advantage of creating webpages that inherited the layout page was that the skeleton of the webpage was already completed and all that needed to be implemented was the content in the centre of the screen that changed from webpage to webpage. This allowed for a more rapid development

of the web apps webpages as significantly less content needed to be provided. This eventually led to the much fast completion of the web application than if template inheritance had not been used.